Advanced Java Programming

Assignment 2

Please comment all classes and methods using appropriate JavaDoc notation.

1. Write a `MyString` wrapper class for an array of characters. Your constructor should accept a character array with the initial data that will be copied into an internal `private` array. `MyString` should provide some of the commonly used methods of the `String` class, and should also override the `toString()` method from the `Object` class to return a `String` representation of the internal character array. Note that the `String` class itself should not be used to implement these methods. Commonly used `String` methods include:

    - `public char charAt(int index)`
    - `public int compareTo(MyString str)`
    - `public MyString concat(MyString str)`
    - `public boolean endsWith(MyString suffix)`
    - `public boolean equals(Object obj)`
    - `public boolean equalsIgnoreCase(MyString str)`
    - `public int indexOf(char ch)`
    - `public int indexOf(MyString str)`
    - `public boolean isEmpty()`
    - `public int lastIndexOf(char ch)`
    - `public int lastIndexOf(MyString str)`
    - `public int length()`
    - `public MyString replace(char oldChar, char newChar)`
    - `public boolean startsWith(MyString prefix)`
    - `public MyString substring(int beginIndex)`
    - `public MyString substring(int beginIndex, int endIndex)`
    - `public MyString toLowerCase()`
    - `public MyString toUpperCase()`
    - `public MyString trim()`

    Implement 5 of the methods above, and try to choose the ones you think will be more challenging. Feel free to implement more. If you don't know what any of these methods is supposed to do, please refer to the documentation for Java's `String` class.

2. Write a `MutableIntArray` wrapper class for an array of integers. The goal of

this wrapper is to simulate an array of mutable length; that is, an array to which we can append and insert values and from which we can remove values. If you are familiar with Java's `ArrayList` class, this is essentially a lightweight implementation of that (if you aren't familiar with it, we will be discussing it soon). You can support this behavior within the class by maintaining an array that is longer than the data you are currently storing. When you need to expand, allocate a new, longer array and copy the data from the old array to the new one. Likewise, you should allocate a smaller array if many items are removed and your array has a lot of empty space. Exactly when to allocate more and less space is up to you. This internal array should be marked `private`.

`MutableIntArray` should have two constructors: one should take an array of integers to supply the initial data to be copied, and another should take a size and internally create an empty array of that size. It should support the methods `public int get(int index)` to retrieve the integer stored at a given index, `public void add(int value)` to append a value to the end of the array, `public void add(int index, int value)` to insert a value at the given index, `public int remove(int index)` to remove a value at a given index and return it, `public int size()` to get the size of the current array, and `public int[] toArray()` to get an array representation of the internal data. Be sure that even if the internal array has empty space, the one returned by `toArray()` does not.

You may not use any data structures other than simple arrays in this assignment, but you may look at the source code for Java's `ArrayList` implementation to guide you.

3. Design an inheritance hierarchy (classes, abstract classes, and interfaces) to reflect a simple version of the animal kingdom. Define and implement methods how and where you think appropriate. Include along with your code an `animals.txt` file explaining your decisions about which classes and methods to implement and how you implemented them. Also feel free to make your taxonomy as expansive as you would like by adding new animals, classifications, and behaviors. In general, species should their own class, and should subclass from more generalized abstract classes that represent taxonomical parents. Behaviors that don't necessarily conform nicely to the taxonomic hierarchy should be abstracted away as interfaces (e.g. flight capability, swimming capability, eating habits (carnivore / herbivore), etc.). In designing your inheritance hierarchy, aim for clarity and simplicity so that the hierarchy feels intuitive for another programmer who might use your classes.